AD-A034 853

LANGUAGE FOR A DESIGN INFORMATION SYSTEM

CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA

AUGUST 1976

LANGUAGE FOR A DESIGN INFORMATION SYSTEM

Charles Eastman

and

Max Henrion

August 1976

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR - TR - 77 - 0014	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) LANGUAGE FOR A DESIGN INFORMATION SYSTEM		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Charles Eastman & Max Henrion		8. CONTRACT OR GRANT NUMBER(s) F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept. Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101D AO 2466
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE August 1976
		13. NUMBER OF PAGES 48
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Air Force Office of SCientific Research (NM) Bolling AFB, DC 20332		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

None

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

LANGUAGE FOR A DESIGN INFORMATION SYSTEM

Charles Eastman

and

Max Henrion

August 1976

This work was supported by the Defense Advanced Research Projects Agency under contract F44620-73-C-0074 and is monitored by the Air Force Office of Scientific Research and by the National Science Foundation under Grant DCR73-03492 A01.

LANGUAGE FOR A DESIGN INFORMATION SYSTEM
CONTENTS:

LANGUAGE FOR A DESIGN INFORMATION SYSTEM

by Charles Eastman and Max Henrion

PART ONE: INTRODUCTION

The authors are part of a group in the Computer Science Department and the Institute of Physical Planning at Carnegie-Mellon University, which is developing a computerised information system for designing large physical systems. The information system is expected to be useful in the design of buildings, ships, mechanical components and other entities that normally require drawings, specifications and a large number of documents for their description and that must respond to a range of diverse analyses. Its purpose is to allow a physical system to be represented within a computer by a single integrated database in sufficient detail for design and construction. It incorporates compact data structures for representing both three- dimensional and other kinds of information. Descriptions of these datastructures have been presented elsewhere [2],[3].

1. OBJECTIVES

The following are some desiderata for such a design information system. A user should be able to define, inspect, modify and analyze different designs, and these operations should be executable in a natural and convenient manner. High-level extensions to the system should be possible for such purposes as automatic detailing, selection and layout of parts, analysis and evaluation. Both interactive use and the application of large, pre-defined programs should be allowed. It may be desirable to interface the integrated database with external analysis packages, which will involve selection and reformatting of data. In addition, it should facilitate convenient output, in such forms as engineering drawings, parts lists and the results of analyses. Many of these requirements may be unique to particular design professions or organizations, and such facilities will have to be capable of being tailored to a variety of professions and organizational environments.

These operations have been conveniently organized as a high level computer language. The objective of this language is to allow the user to easily perform the operations required to design within the computer database. In this sense, it is a LANGUAGE FOR DESIGNING. We have developed and are implementing such a language, named GLIDE (Graphical Language for Interactive DEsign). In this paper, we describe the semantics and associated syntax of this language.

The basic unit of information within GLIDE is an ELEMENT, defined as a set of attributes describing some coherent entity. An Element may be a building component, a space or activity area, building subsystem, or any other entity relevant to design. ATTRIBUTES are properties of an Element, including both spatial ones, such as location and shape, and others such as weight, material, surface and color. In GLIDE, a shape is defined as a planar polyhedron. Attributes may also represent relationships between the referent and one or more other Elements. Some attributes are system defined and maintained while others can be defined and manipulated by the user.

Many design components are standard units and described in catalogs. Many abstract entities, such as activity areas in building design, are also well defined and available from handbooks. All such standard information used in design should be accessible from stored catalogs. Other elements are unique and defined by a designer. There should be convenient graphical means to enter such information into the system.

As elements are included, they are arranged in spatial relationships. It is often desirable to operate on a whole collection of elements as a unit. A language for design should allow the formation of sets of elements and operations on those sets.

Design tasks vary from the very repetitive generation of standardized plans to the creative and sometimes idiosyncratic development of solutions to unique problems. The full set of objectives involved in a design are rarely achieved on the first pass. Several iterations are usually required, some for exploring alternative configurations, and others for considering alternative objectives, which often change as the design evolves. Moreover, a completely detailed description of the sub-systems making up a design is often not available at the start. Rather, the properties of the various subsystems are defined incrementally, in a particular evolutionary sequence. This might be called a DESIGN SEQUENCE. Most CAD systems developed to date incorporate a rigid design sequence, which is effective for only a very limited range of conventionalized results. A language for design should be able to accommodate a range of different kinds of design sequences reflecting different styles of problem solving.

In summary, then, a language for design must minimally support:

1. The selection of elements from a catalog,

2. The graphical definition of custom elements,

3. The placement and grouping of elements, individually or in sets.

4. Visual inspection of a design, in perspective, orthographic, or schematic representations.

5. Procedures for analysis and evaluation of a design, eg. structural, acoustical, thermal,etc., either as packages within the language or by providing interfaces for passing the required data to existing analysis programs.

6. Evaluation of the internal consistency of a design, with regard spatial conflicts, joining conditions,etc.

7. Alternative top-down, bottom-up or other evolutionary sequences of design.

Some other requirements will be introduced as we proceed.

2. LEVEL OF THE LANGUAGE

The development of a computer language for design has no heritage on which to stand. It is generally agreed that there is no natural language of design currently in widespread use, in contrast to, say, medicine or law. It seems doubtful that any one set of high level structures and operations could be defined that, alone, would be sufficient for a general range of applications.

The alternative is to provide a set of basic commands, plus general language features, to allow the definition of higher level commands for particular applications. These facilities should provide an environment in which a design organization can easily develop software to support its own particular design sequences and methods. Thus the language should consist of a set of basic commands for conveniently creating, displaying and modifying design Elements with powerful features for dealing with geometric properties. In addition it should contain general control structures and means for defining new procedures and record formats, which can be used to extend the basic set of operations and data structures.

The user of the command language and its extensions would be a designer, architect or other person without programming expertise. However the user of the complete language, who will write these extensions, would probably be a specialist applications programmer. In practice there need be no hard dividing line between these two levels of GLIDE and learning of the complete language should be feasible in small incremental steps from the simpler sub-language.

For direct interaction with the database the operating environment should be interpretive, ie commands should be translated and executed as they are entered line by line. But for efficient execution of pre-defined procedures it is desirable that they are compiled once only, at the time they are defined. It is important that such pre-compiled procedures be independent of the particular state of the database at compilation time, so that their validity cannot be affected by any changes to it. Hence it must be possible to delay binding or linking of procedures to other procedures and entities in the database until execution.

3. ISSUES OF IMPLEMENTATION

Large database systems can be implemented in two basically different ways: as an extension to an existing language, such as FORTRAN, or as an independent but complete language unto itself. GLIDE places heavy emphasis on the manipulation of new, large, system defined records and the operations on these records were among the major concerns of the language design. Graphical interaction is to be the major form of interaction and extensions are required for even this type of input-output. GLIDE is also oriented toward mini-computer implementation. The range of high level languages to extend was thus quite limited. Probably most persuasive, however, was the recognition that GLIDE is a research vehicle, in which we expect to study the operators and syntax most desireable for developing high level design applications. Restrictions on the style of language, imposed from the base language adopted, would inhibit work toward this objective. For these reasons, we chose to implement GLIDE as an essentially new language. This, however, did not preclude us from adopting a standard Algol-like syntax where appropriate.

A major feature of GLIDE is its focus on spatial information and its reliance on interactive graphic techniques for editing it. In the language definition, however, we have taken a conservative stance toward graphical operations and have defined only the minimal level needed for this end. They are easily implemented in direct view storage tube terminals. Later, we may propose extensions that will provide more powerful graphical operations.

GLIDE utilizes the spatial information record scheme developed in BDS[2],[3]. That is, it stores shape topologies separately from geometries and requires a shape to reference a topology by name. Topologies can be created by using special operators, as well as convenient graphical methods. Other spatial representations are compatible with GLIDE, however, and we have attempted to provide a syntax and set of commands that are inclusive of a range of spatial representations. Shapes are manipulated using the spatial union, intersection and complement operations.

4. METHODS OF REPRESENTATION

An important question in designing representations is whether the information should be embedded in procedures in implicit form rather than be represented as explicit data. A procedural representation requires less storage space, but at the cost of extra computation every time the data is required. This compute versus store question is not a dichotomous choice but a gradation of possible trade-offs. The best solution for a particular application depends on both the computing resources available and the actual demands upon the system.

At one level this trade-off is a question of the database implementation and is more or less independent of the language design. For example, in BDS the vertices of each shape (Form) are stored only once, and the actual co-ordinates of each instance (Copy) in space are computed only when needed. This and other decisions were taken to optimize available computing resources, on the basis of an analysis of expected demands[1]. However GLIDE could well be implemented with a quite different database organisation.

At a higher level each segment of Glide code constitutes a procedural representation of some information. In general, source code is the most compact way of representing a database. We anticipate that a catalog of design elements will be built up as a library of procedures. In an interpretive environment it is up to the user to control when a procedure should be converted into explicit data and how long this structure should be retained. Some features of the language aid in this task. Any records declared locally within a procedure are automatically deleted at the end of execution. Thus the information is incarnated only for the duration of execution of the procedure. Alternatively the new records may be declared as global within the procedure, in which case they persist in the database until they are deliberately deleted. Any further calls to that procedure do not have to recreate the records but can use the pre-existing definitions. For many practical purposes, such as detecting spatial conflicts between objects, a procedural representation would be very unwieldy for large databases. Therefore we anticipate that during interactive design the primary method of representation will be as explicit data.

5. SYNTACTIC STYLE

Before we go on to describe a particular instantiation of these language features, some general issues affecting the structure and syntax of the language should be mentioned. While the selection of many semantic features flows fairly directly from the needs of a design information system, the choice of syntactic forms will tend to be more controversial. It seems evident that syntax affects the ease of learning and of use of a language, especially for users with little or no programming experience. For more discussion of these issues see Wirth [4]. One day experimental evidence of the relative merits of syntactic features may be obtained by comparison of alternative implementations. Meanwhile the priorities to be awarded to these criteria must be based on conjectures about future use and users and the relative efficiency of alternative implementations.

GLIDE syntax reflects the twin objectives of providing both a simple command language and a more general language, with features allowing extensibility. For the first objective, procedures may be called with a "command-style" syntax; that is, the arguments need not be enclosed in parentheses and alphanumeric mnemonics may be used as separators instead of commas. The more conventional function notation calling syntax may also be used when preferred. Frequently used constructions such as those for defining a location and orientation in space, or the co-ordinates of a shape vertex, have special concise forms with convenient defaults. It also happens that the reduction of redundancy in the database, especially with regard to spatial information, encourages conciseness in the linguistic forms for entering the information, and vice versa. Thus the task of entering spatial information is much eased, and of course it can be further eased by the use graphic techniques.

The full language incorporates the usual features for control structures and definition of procedures. For these we have adopted an Algol-based syntax, with standard block structure. We have followed the precepts of structured programming. GO TOs are not provided. GLIDE is expression-oriented. That is, most statements return a value and can be treated as expressions. For example BEGIN END blocks return the value of their last statement.

In addition to the simple data types - number, boolean and text - GLIDE has a number of record types which provide special structures for representing element information. These include TOPOLOGY as mentioned above, FORM which defines a class of elements and is akin to a user-defined record format, COPY, being an instance of a Form, and SET, being a group of elements. As in ALGOL, all identifiers must be declared and their type specified before they can be used. In most cases the data type of an operand or argument is determined by its context and can be checked at compilation. Block structure allows dynamic declarations and allocation of memory during execution, as in Algol or PL/1.

Since a prime objective is the creation of new entities forming a database, there is particular emphasis on declaration features. Notably, global records can be created from within inner blocks including from inside procedures. Correspondingly, deletions are also allowed.

6. CONCLUSIONS

The second part of the paper gives a preliminary description of a language which attempts to fulfill the requirements outlined above. In the third section we present some example programs, with which we have attempted to indicate the kinds of design operations easily defined within GLIDE. The full extent of the strengths and weaknesses of the language will only be known after extensive experience with it and other design oriented languages.

This work was funded by the National Science Foundation. We have received many useful suggestions throughout the development of this work from many collegues in architecture and computer science, especially Adrian Baer, Gilles Lafue, Joseph Lividini and Bob Kenyon.

PART TWO: A SPECIFICATION FOR GLIDE

1 FUNDAMENTALS

1.1 OPERATING ENVIRONMENT

The operating environment for Glide is semi-interpretive. In general, each statement typed into the system is immediately translated and executed. It is also possible to combine a number of statements into a single block-statement enclosed between BEGIN and END. In this case translation, and therefore syntax-checking, takes place statement by statement, but execution is delayed until the END of the block. After each line the system responds with a prompt character and a number of tabs equal to the depth of block nesting, providing automatic indentation reflecting block structure.

Programs to be stored and executed repeatedly are defined as procedures. They are compiled as they are first entered into the system, statement by statement, with the resulting syntax check. A routine library contains predefined procedures for various data-base manipulations and standard design operations. It can also contain routines to create standard design components (Elements) and thus serve as a parts catalog.

1.2 LEXEMES

There are four kinds of lexeme or terminal symbol:

1 Special syntax symbols, such as + * [ ] ( ) # , :, each being a single non-alphanumeric character. Listed in appendix 3.
2 Keywords such as BEGIN END FOR IF. Listed in appendix 3.
3 Literal numbers, both integer and real. See section 2.1.
4 User-declared names or identifiers.

User-declared names consist of a letter or "$" followed by zero or more letters, digits, "." or "$". Only the first 31 characters of a name are significant. Upper and lower case letters are treated as equivalent throughout.

Successive alphanumeric lexemes must be seperated by a space, tab or new-line, and so these characters cannot occur within them. Beyond this, space, tab and new-line can be used freely anywhere in Glide code and are ignored by the interpreter.

NB <n> denotes examples of Glide code here and subsequently.

```
<> Legal names:
A, PDP11, U.S.A., C.mmp, Spiral.staircase.15,
etc..., $Mellon
Illegal names:
.NAME, 4B, Go away, PDP-10

letter ::= A|B|C|...|Z|a|b|c|...|z|$
digit ::= 1|2|3|4|5|6|7|8|9|0
name ::= letter | name letter | name digit | name .
```

NB: See appendix 1 for details of the BNF conventions used.

1.3 DATA TYPES

All variables and expressions are of a particular data type. Variable names must have their type declared before they can be used. In most cases the context in which a variable occurs in a statement determines what type it should be and the interpreter will automatically check that it is correct.

There are three simple data types - Real, Boolean and Text. They consist of a single value, in contrast to records which can contain several simple values or other sub-records. Record types include Topology, Form, CopySet and View. Vectors, or 1-dimensional arrays also are provided, but can only be of simple types.

ELEMENTS in the data-base represent such things as physical objects and spaces. An Element consists of a set of ATTRIBUTES, which may include both user-defined Attributes of the above data types, and some system-structured Attributes such as SHAPE and LOCATION. FORMS are exemplars or prototypes for a class of similar Elements. All Elements in such a class are COPIES of the Form from which they are derived. Forms may also be considered as user-defined record types, of which Copies are instantiations. SETS are collections of Elements. ITEMS are reference variables which refer to Elements and Sets. TOPOLOGIES are used to help define the Shape Attribute of Forms. The names of these four record types are bound to their referents at definition, and the binding is fixed for as long as they exist. Such names are constants rather than variables. Therefore two additional types are provided which are reference variables: ELEMENT refers to Copies and ITEM can refer to Copies or Sets.

In addition the Geometric types - FACE, EDGE and VERTEX - are parts of a Shape. The exact nature and use of all these types will be described later.

```
simple-type ::= REAL | BOOL | TEXT
record-type ::= TOPO | FORM | ITEM | SET | VIEW
geom-type ::=  FACE | EDGE | VERTEX
type ::= simple-type | record-type
```

1.4 STATEMENTS AND EXPRESSIONS

GLIDE is an expression oriented language. Most statements evaluate to a particular value of known type. The type and value of the different kinds of statement will be defined as we proceed. Conversely expressions, such as arithmetic expressions or record definitions, can be considered as statements in many contexts. There are six basic kinds of statements in addition to expressions - control statements, procedure calls, assignments, comments, declarations and blocks. Expressions will be described under headings for their type. The control statements, including the conventional conditional and loop statements are described in section 6. The remaining kinds of statement are described below.

Infix operators are defined for various data types, including the arithmetic operators (+-*/↑), boolean connectives, assignment (←) and others. The full precedence hierarchy for all infix operators is given in full in Appendix 2. Higher precedence means those operators are evaluated first. Equal precedence operators are evaluated from left to right. The system precedence ordering can always be overridden by putting parentheses around those subexpressions which are to be evaluated first.

```
statement ::= expression | control-statement | proced-call
            | assignment | comment | block | declaration
expression ::= n | bool | text | topo | form | elem | item
            | set | view | "("expression")"
```

1.4.1 PROCEDURE CALLS

Procedure calls can use standard function syntax, with the procedure name followed by arguments, which are enclosed in parentheses and seperated by commas. It is also permissable to omit the parentheses. Where they have been declared, alphanumeric symbols may be used as seperators instead of commas. This allows a "command-style" syntax, which may be easier for non-specialist users.

```
<>
INVERT(a,b,c)
SIN(x)
ALIGN a WITH b ALONG x
REPEAT beam BETWEEN x1,y1,z1 TO x2,y2,z2 TIMES 10

proced-call =  proced-id [( arg {sep arg} )]
             | proced-id [arg {sep arg}]
arg = expr
sep = name | ,
```

1.4.2 ASSIGNMENT

The operator "←" assigns the value of an expression into the variable, which may be a simple type or item variable. The expression to be assigned is checked for type compatibility with the variable to which it is assigned. Note that since record names are fixed names and not variable it is meaningless (and illegal) to assign to them. The value of an assignment statement is the right operand. Thus assignments may be chained:

```
<> A←B←C←0;

assignment ::= num-id[index] ← n | bool-id[index] ← bool
     | text-id[index] ← text | item-id ← item | elem-id ← elem
index ::= "[" n "]"
```

1.4.3 COMMENTS

Syntactically, comments are statements which start with "!". They may contain any character except ";" since this terminates them. They are ignored by the interpreter.

```
comment ::= ! any-chars-except-;
```

1.4.4 BLOCKS

Blocks consist of a sequence of statements seperated by ";"s and enclosed between BEGIN and END. Since a block is also a statement, this definition is recursive and allows blocks to be nested inside one another.

A block can also be treated as an expression and it has the value and type of the last statement in it.

```
block ::= BEGIN statement {; statement } END
```

1.4.5 BASIC DECLARATIONS

All names (identifiers) must have their type declared before they can be used. The general format for declarations is the type followed by a list of names seperated by commas. When a record name is declared it may be bound to a definition in the same statement using the operator "=". Basic declarations return no value, but declarations binding a name to a record definition have that new record as value and type.

```
<>REAL x,y,z;
ITEM top,bottom, side.piece;
SET boxes = {matchbox[1]; shoebox[3]; glovebox[2] }

basic-decln ::= simple-decln | record-decln
simple-decln ::= simple-type name {,name}
array-decln   ::=   simple-type   VECTOR   name   [size]   {,name[size]}
size ::= "[" n "]"
record-decln ::= record-type name {,name}
record-binding ::= TOPO name = topo | FORM name = form
        | SET name = set | VIEW name = view
```

1.5 BLOCK STRUCTURE AND THE SCOPE OF NAMES

As in Algol 60 and its derivatives, programs in Glide have a block structure which defines a hierarchy of scopes for local variables declared within them. Unless specifically indicated otherwise, variables are local to the block in which they are declared. This includes simple, vector and reference variables, but not record names, which are always global wherever they are declared. For local declarations, the scope within which any name may be referenced is limited to the block in which it was declared and any sub-blocks it may contain. Two variables with the same name may be declared at different block levels. In that case the name is assumed to refer to the variable declared at the innermost block level in whose scope it occurs.

An entire design project may last over many sessions at the terminal. Conceptually the scope of a project corresponds with the outermost block level. This outer block begins at the initiation of a project and is the outer block level during a terminal session. Therefore variables and entities declared at this top level are essentially GLOBAL and continue to exist between sessions over the length of the project.

Record names have global scope no matter where they are declared starting from the time at which their declaration is first executed. Records defined in this way persist as a permanent part of the database unless they are explicitly deleted. Nevertheless it is also possible to create temporary records if they are not given a fixed name but created within a set or assigned to a reference variable. In this case their scope is that of their longest-lived reference. This allows, for example, procedural representations of objects, which have only temporary existence as data during the execution of the procedures.

Procedures are compiled when they are defined, but may be executed much later. Thus compilation should be independent of any particular globals in the data base. the linking of references within the procedure to pre-existing globals is done at run-time. All such globals mentioned must be previously declared within the procedure as external. Such declarations are introduced by the keyword GET.

So far, with the exception of the interpretive environment, we have described the conventional scope hierarchy. But Glide also contains some additional features:

1 Declarations are not restricted to occurring at the beginning of a block but may occur wherever a statement may occur.

2 Global declarations of variables also can occur at inside block levels. This is indicated by using the keyword GLOBAL before the declaration. In the case that a record name of a variable declared as global has been previously similarly declared, then the current declaration is treated as an external declaration. That is the existing name and any value or definition it may have is linked to the current segment of code at execution time. If the current declaration is of a record type with a binding to a definition, this new definition is ignored. This feature allows procedures to create Global records they may need the first time they are called, and then use them on subsequent calls without having to recreate them. In this way it is similar to the "own" variable concept, but it can be accessed from outside the block in which it was declared, after the first execution of that procedure.

3 The scope of variables can be limited within their natural scope by deleting them explicitly with the DELETE command. This is particularly useful for removing globals that are no longer required.

It should be noted that GET, GLOBAL and DELETE statements are all executed at run-time and hence they can have no effect on the data-base at compilation.

```
<>SET DRAINS = { pipe[1,3,5]; drain[1 TO 15]}
GET FORM tree, bush; GET ATTRIB TEXT color, density;
DELETE west.wing, east.wing, plumbing;

declaration ::= basic-decln | attrib-decln | record-binding |
                global-decln | external-decln | deletion

global-decln    ::= GLOBAL basic-decln | record-binding
deletion ::= DELETE name {, name}
external-decln ::= GET basic-decln | extern-proced-decln
```

2 SIMPLE TYPES

2.1 NUMBERS

All numbers are treated as real (floating point), although both integer and real syntax is acceptable for constants (literals). Numerical expressions are created using the usual arithmetic operators: addition, subtraction, multiplication, division and exponentiation. (+ - * / ↑). ↑ has highest precedence, above * and / , which are above + and - .

```
<> REAL x,y,z; z←125.2+100/(a↑2-b↑2);

n ::= num-id[index] | attribute | proced-call
      | number-literal | num-expr | statement
num-expr ::= n arith-op n | ( n )
arith-op ::= *|/|-|+|↑

integer-literal ::= digit{digit}
real-literal ::= [integer-literal] . integer-literal
number-literal ::= [-]real-literal | [-]integer-literal
```

2.2 BOOLEANS

The boolean constants (literals) are simply TRUE and FALSE. The relation operators compute boolean values from pairs of numbers. The boolean operators AND and OR compute boolean values from boolean operands. Only the first operand is evaluated if this is enough to determine the value of the expression, eg in:

```
b1 OR b2
```

b2 is not evaluated if b1 evaluates to true.

```
b ::= bool-id[index] | attribute | proced-call
      | bool-lit |   bool-expr | statement
bool-expr ::= b bool-op b | n rel-op n | NOT b
      | ( b ) | expr = expr
bool-op ::= AND | OR
rel-op ::= LSS | LEQ | EQL | GEQ | GTR | NEQ
bool-lit ::= TRUE | FALSE
```

2.3 TEXT

Text variables take as values character strings of arbitrary length. Text literals are indicated by enclosure in single quotes. If a literal ' is desired in a string, this is indicated by two consecutive ones: ''.

```
<>
TEXT NAME,TAG
NAME←'FRANZ KAFKA'; TAG←'DON''T FORGET'
```

text ::= text-id[index] | attribute | proced-call | text-literal | statement
text-literal ::= ' char-string-with-no-single-' '

2.4 VECTORS

Variables of any simple type may be declared as vectors with bounds from 1 to n where n is given in square brackets following the name in the declaration statement. An element of a vector is indicated by a subscript in square brackets, when it can be used in an expression. Arrays of more than 1 dimension are not provided.

```
<>
TEXT VECTOR days[7],months[12];
BOOL VECTOR gridx[40],gridy[40];
gridx[i]←gridx[i*3] OR gridy[i*3+1];
```

3 ELEMENTS

Representations of physical objects and spaces in the data base are known as ELEMENTS. Each Element is defined by a set of ATTRIBUTES which may include both spatial Attributes such as Shape and location and non-spatial ones such as cost, density, color, thermal conductivity etc. Elements can also represent abstract entities, such as design goals, and need not have any spatial Attributes.

One way of defining an Element is to enter its entire complement of Attributes. This is known as a FORM. The other way of defining a new Element is to describe it as a COPY of an existing Form. It often happens that classes of Elements exist that have many Attributes in common, especially their Shape. In this case it is easier to define an Element by exception, in terms of its differences from an existing Element. One Attribute certain to vary between Elements is the location. A Copy is defined by entering its location and any other of its Attributes that differ from the Form from which it is derived. The Shape of a Copy is largely determined by its originating Form, but may vary in certain ways described in Section 3.8.3. The Form can be viewed as a typical exemplar for a class of related Elements. This organization allows a convenient means for describing many similar objects.

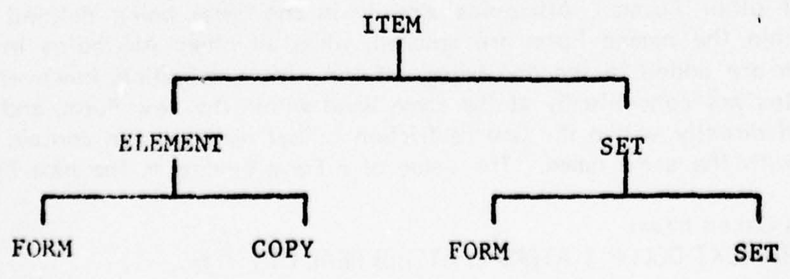
3.1 NAMES OF ELEMENTS

A Copy is identified by adding a non-zero subscript to the name of the Form from which it is derived. The zeroth Copy is the same as the Form Element.

The name of a Form or Copy is bound to a single general description throughout its life. While part of the definition can be modified it is not possible to substitute a new definition. In contrast, a name declared as an ELEMENT can refer to a Form or Copy record. The Element variable can be made to refer to a new record simply by reassigning it. Element is the union of Form and Copy.

Similarly a name declared as an ITEM is a reference variable to an Element or a Set of Elements. (Sets are described in section 4.) Item is the data type which is the union of Element and Set. It is useful for operations and procedures which can operate on either single Elements or collections of them.

These types can be organized in a hierarchical fashion, as shown below. Each type can refer to any of those below it in the hierarchy:



```
<> ELEMENT thing;
thing←box[10];
ITEM this, that, the.other;

copy ::= form-id[index]
elem ::= elem-id | copy | form | proced-call | attribute
item ::= item-id | elem | set
```

3.2 ATTRIBUTE DECLARATION

An Attribute consists of a name and a value. The same Attribute name, eg. COLOR or DENSITY, may be relevant to many different Elements, and hence Attribute names are always Global. Some Attributes such as Shape and Location are system defined. Further Attributes may be declared by the user to be of any variable type including vectors, and references to records. The Attribute declaration is introduced by the keyword ATTRIB, but otherwise resembles a normal declaration. Attribute declarations return no value.

```
<1> ATTRIB TEXT COLOR, MANUF, MATERIAL;
ATTRIB ITEM PARENT,CHILDREN,SIBLINGS;
ATTRIB REAL VECTOR CofG[3], LENGTH[4];

attribute-decln ::= ATTRIB basic-decln
```

Both system and user defined attribute names identify fields within Element records. Elements are defined by assigning values to these fields, either within the Form definition itself or, as exceptions, within the Copy record. All Copies of a Form have the same set of Attributes.

3.3 FORM DEFINITIONS

Each Element is defined through a Form or Copy declaration. A Form is defined by a Form procedure or a Form definition block. A Form definition is enclosed by the brackets, BFORM and END or { and }, which are equivalent. It contains a set of one or more Attribute assignments, seperated by ";"s. Each assignment consists of an Attribute name which defines a field of the Form record of a given type, and an expression with matching type, whose value is used to initialise the field. The Form definition can also contain an existing Form name or procedure call that returns a Form. The Attributes defined in these are added to the current Form and thus Forms can be defined hierarchically in terms of other Forms. Attributes already in the Form being defined and existing within the named Form are ignored, while all other Attributes in the named Form are added to the one being defined. After definition, however, all the Attributes are conceptually at the same level within the new Form, and can be accessed directly within it. One restriction is that no Form can contain two Attributes with the same name. The value of a Form binding is the new Form.

```
<> An Oaken beam:
ATTRIB TEXT COLOR, MATERIAL; ATTRIB REAL DENSITY;
```

```
FORM BEAM = BFORM
        MATERIAL←'Oak';
        DENSITY←0.532;
        CUBOID( 2.5,1,3);                    !procedure call defining a Shape.
        END;
FORM NEWBEAM = {COLOR←'white'; BEAM}

<> A specification for an office work area for one person:
        ATTRIB TEXT TITLE, NOTES;
        ATTRIB REAL AREA, SHELFLENGTH, DESKFT, FILESPACE;
        ATTRIB BOOL PHONE;
FORM WORKSPACE =
        {TITLE←'secretary';
        AREA←25; SHELFLENGTH←15; DESKFT←5; FILESPACE←3.25;
        PHONE←TRUE; NOTES←'Likes green wallpaper' };
```

```
form-binding ::= FORM name = form-defn
form = form-id | proced-call | form-defn
form-defn ::= "{" statement {; statement} "}"
            | BFORM statement {; statement} END
```

A statement in the context of a form definition block can be an attribute assignment, or have the latter embedded within them.

```
attrib-assign ::= attrib ← expr | shape-spec | statement | form
```

3.4 LOCATION

Location is a system defined Attribute that must be specified for each Copy of a Form. It may be specified in world co-ordinates or relative to the origin of any existing Element. This location is defined by 6 numbers - denoting the X, Y and Z co-ordinates and rotations in degrees round these axes. The rotation is performed successively round the X, Y and Z axes in that order, prior to the translation of the Shape. These fields within the Location Attribute can be viewed as subattributes and have system defined names for accessing them. They are:

CX,CY,CZ	the three translation values defining an offset.
AX,AY,AZ	the three rotation values defining a orientation.

Locations will be entered very frequently and so their syntax has been designed with conciseness in mind. A location may be specified as relative to an existing item, introduced by "@" or, if this is omitted, as relative to the co-ordinate origin (absolute.) The offset and rotation are each specified by up to 3 numbers, introduced by "\" and "#" respectively. Either or both may be omitted. Any numbers omitted default to zero.

```
<>
\10,20,30 #90,0,90
\10 #90,45 is equivalent to \10,0,0 #90,45,0
```

```
\25,20 is equivalent to \25,20,0 #0,0,0
#180,90 is equivalent to \0,0,0 #180,90,0
@ MATCHBOX[5]#0,20 is a location relative to MATCHBOX[5]

location ::= [@ copy][offset][rotation]
offset ::= \ triad
rotation ::= # triad
triad ::= n [,n [,n]]
```

3.5 COPY DEFINITION

A Copy is named by giving an existing Form with a new subscript. The subscript may be specified by the user as any integer not yet used. If it is omitted the system supplies in default the integer succeeding the current highest subscript. The Copy definition is enclosed between { and }, and its first part specifies a location. The value of a Copy definition is the Element defined.

```
<> COPY BOX[1] = {\20,30,100};
COPY BOX = {\100 #0,180};
COPY BOX = {@ WINDOW[2] #180,90};
```

The Form defines the allowed set of Copy Attributes and their default values. For all but the Shape these values may be overwritten within each Copy. In this case the Copy definition includes the appropriate Attribute assignments after the location specification.

```
<>COPY BOX = {\10,10 #0,90; COLOR←'black'; LENGTH←2.2};

<> A procedure to create a row of Copies at intervals.
For the syntax of procedures definitions, see Section 7.
PROCEDURE REPEAT (FORM f; :FROM REAL x,y,z :BY dx,dy,dz :TIMES n)=
      FOR i FROM 1 TO n DO COPY f ={\x+dx*i, y+dy*i, z+dz*i};
! Use procedure to make a stack of 20 boxes.
REPEAT box FROM 100,200,0 BY 0,0,20 TIMES 20;

copy-decln ::= COPY form [index] = copy-defn
copy-defn ::= "{" location {; attrib-assign } "}"
```

3.6 ACCESSING ATTRIBUTES

Attribute values may be accessed but not changed outside of the Form or Copy definition. They may be used in expressions as a subexpression of the appropriate type, but they may not receive assignments. The general syntax for an Element Attribute is:

```
attribute ::= attrib-id[index] OF copy

<> shade← color OF box[1];
```

If several Attributes are to be accessed from a particular Form or Copy, they may be put inside a Form or Copy definition block, which has the effect of defaulting the Form or Copy name for all Attributes named within it. The default Element can be overridden, however, by using the full Attribute name, eg.

```
<> BOX[1]{length←heighth+length OF box[0]};
```

3.7 MODIFYING AN ELEMENT RECORD

A Copy or Form can be modified using a syntax similar to that by which it was initially defined. New attributes can be added to a Form by using the Form or Copy block structure. Note that certain Attributes, such as Shape and Vertex co-ordinates (see section 3.8), that were themselves bound in their initial definition cannot be subseqently modified.

A new value may be assigned to an Attribute of a Copy even if it was not originally defined as different from the Form value, however it may not be given an Attribute not hitherto defined for the Form.

Note that assigning a new Attribute value directly to a Form will change all Copies with the default value for that Attribute, and hence must be used with care.

```
<>ATTRIB TEXT material;
FORM table = {material←'wood'; color←'brown';
              tabloid(1,4,0.5) }
COPY table[1] = {\250,350; material←'steel'}
table[1] = {color←'white'};
ATTRIB REAL cost; table = {cost←75 };
table[1] = {cost←100}
```

3.8 SHAPE

Shape is an Attribute of particular importance for designing physical systems. Shapes are represented as polyhedra with planar faces. The information required to describe a shape is divided into two parts: The TOPOLOGY and the GEOMETRY. The TOPOLOGY describes the adjacency relations between the Faces, Edges and Vertices of the polyhedron. The GEOMETRY specifies the relative position of the Faces, Edges and Vertices. Thus a Topology formats the Geometry information. Topologies may be common to many different Shapes and can be entered independently from any particular one. (See figure 1)

The Shape Attribute is associated with a Form. It is one Attribute that may not vary in the Copies of a Form. It consists of a number of subrecords which include FACES, EDGES and VERTICES. A Topology defines the number and organization of these subrecords for each Shape. Each Face consists of an ordered set of Edges. Each Edge contains two Vertices. The Faces, Edges and Vertices are uniquely numbered within each Shape and this index is used to identify them.

3.8.1 TOPOLOGY

A Topology is a record type that is defined independently of the Elements it partially describes. A new Topology may be constructed by means of Euler operators, which combine new Vertices, Edges and Faces. (They are named after Euler who showed they are sufficient to construct any legal polyhedron.)

In order to construct a Topology, a user must declare a set of Faces, Edges and Vertices from which to construct it. These records are global and always available within a Topology definition block. When used to define a particular Topology, however, they are bound to that definition and only accessible through the Topology record they are bound to. They are declared using:

FACE[n]; EDGE[m]; VERT[p];

where the n,m, and p denote the number of new records of each type to be created. As these are used in making Topologies, the remaining are renumbered in consecutive order. If new ones are added, these are added to the end of those already existing.

The primitive operators for constructing Topologies from these records are:

MVE(v1,v2,f)	Make a new Vertex v2 linked to v1 on Face f.
MFE(v1,v2,f1,f2)	Make a new Face f1 by linking v1 and v2 on f2.

Topology subrecords must be assigned in such a manner that the final product contains Faces and Vertices numbered successively from 1 to the total number. To assist in this the following operators supply consecutive numbers for new faces and vertices.

CVE(v1)	Create a new Vertex linked to V1, and supply new vertex number
CFE(v1,v2)	Link vertices v1 and v2, and supply new face number.
CMV(n,v1)	Create Multiple Vertices in a chain of n from v1.

It is possible to create a new topology by modifying an existing one. A copy of an existing one can be set up at the start of a Topology definition block using:

FETCH topo	Copies the named topology to be modified.

A new topology is constructed inside a topology-definition block, delimited by BTOPO and END. (As in all record definitions { and } may also be used as delimiters.) Only within this block can the Euler operators be accessed.

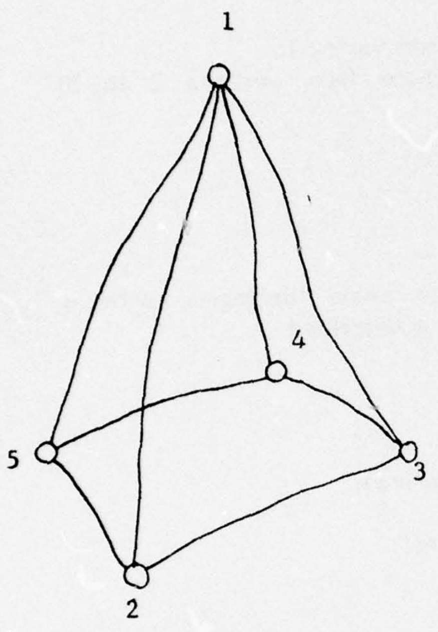
Other statements can be mixed with them. The value of a Topology definition is the record created. Some examples of Topology definition follow:

```
<>TOPO PYRAMID = BTOPO
        FACE[5]; EDGE[8]; VERT[5];
        CMV(4,1);       !a chain of 4 vertices from vertex 1;
        CFE(2,5);               ! Create the base face, vertices 2 to 5;
        CFE(1,3);       !vertex 1 is the apex;
        CFE(1,4);
        CFE(1,5)
        END;
```

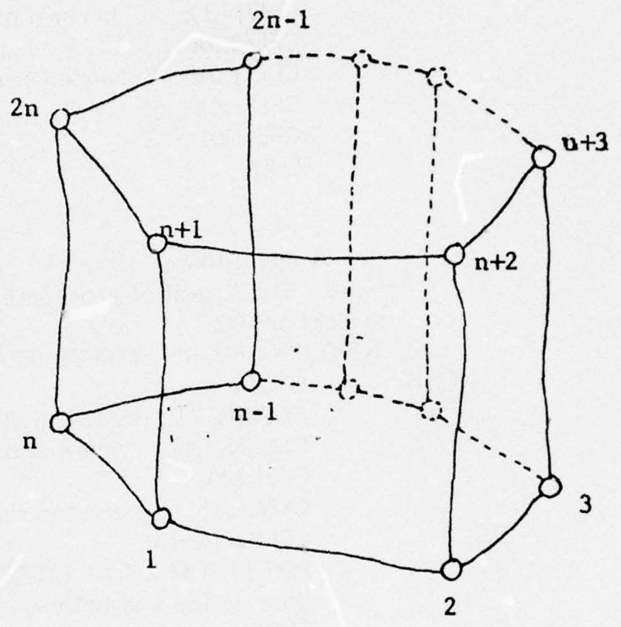
<> A procedure to make extrusions or prism topologies, with n sides. The format of procedure declarations is described in Section 7.0.

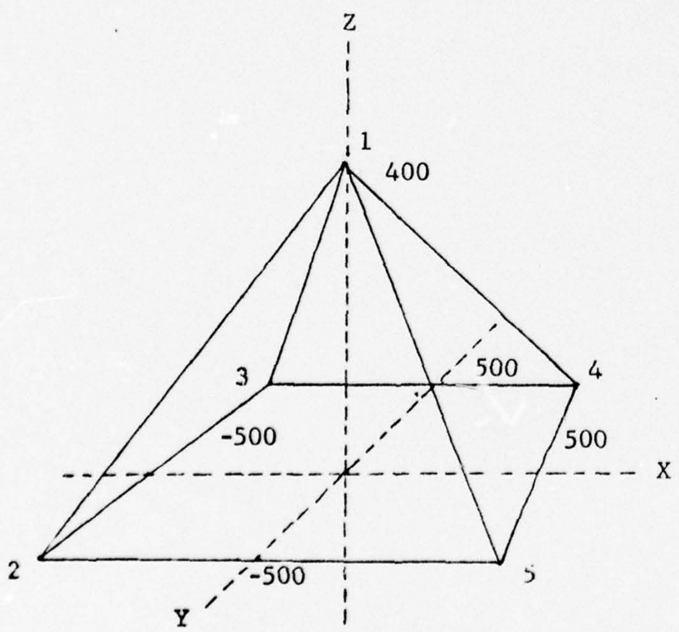
```
TOPO PROCEDURE EXTRUDE (REAL N) =
        BTOPO
        FACE[N+2]; EDGE[3*N]; VERT[2*N];
        CMV(N-1,1);     !the bottom ring of n vertices;
        CFE(N,1);
        CMV(N,1);       !the top ring of n vertices;
        CFE(N+1,2*N);
        FOR I FROM 2 TO N DO CFE (I,I+N)
        !join up top and bottom;
        END;
```

Pyramid Topology

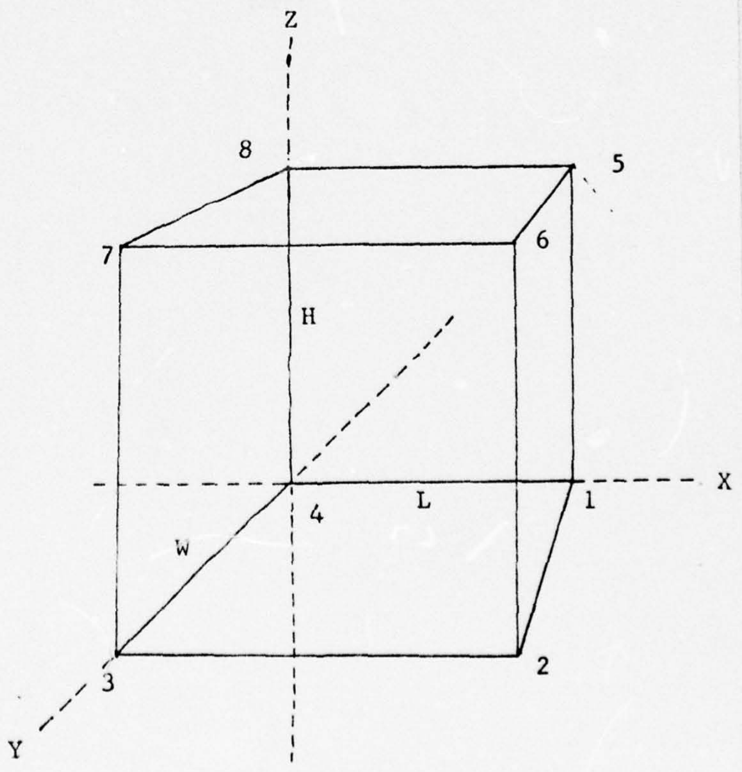


Extrude Topology





Cheops Pyramid Shape



Cuboid Form

```
geom-decln ::= geom-type index
( geom-type ::= FACE | EDGE | VERTEX
index ::= "["n"]"
topo-decln ::= TOPO name = topo
topo ::= topo-id | attribute | proced-call | topo-definition
topo-definition ::= BTOPO {geom-decln;} topo-stmt {;
      topo-stmt } END
topo-stmt ::= euler-op | statement
```

Any statement in the context of a Topology definition can contain Euler operations embedded within it.

```
euler-op ::= CFE n,n | CVE n | MFE n,n,n | MVE n,n,n,n | CMV n,n
```

In addition it is possible to enter topologies graphically on a digitizing tablet, representing them as a planar graph. Control is passed to the digitizing program by a procedure DRAWTOPO. When entry is finished this procedure returns as its value, the new topology record.

```
<> TOPO NEWSHAPE = DRAWTOPO;
```

3.8.2 TOPOLOGY PRIMITIVES

There is a need to access Shape information through a Topology in certain applications. See for some examples Part Three, Section 3. The following functions are available for accessing topogical relations within a Shape.

E.FACE(f,a)
returns the Edge (index) following Edge "a" on Face f. If a=0 then it returns the first Edge.

V.FACE(f,a)
returns the Vertex following Vertex "a" on Face f.

F.EDGE(e,a)
returns a Face adjacent to Edge e; one of the two is returned if a=0, the other if a=1.

V.EDGE(e,a)
returns a Vertex on Edge e; one if a=0, the other if a=1.

F.VERT(v,a)
returns the next Face after "a" adjacent to Vertex v.

E.VERT(v,a)
returns the next Edge after "a" adjacent to Vertex v.

Each operation returns the index of the geometric primitive identified.

3.8.3 GEOMETRY

A Shape is composed of the Face, Edge, and Vertex subrecords. These subrecords also have system defined subattributes. These attributes take as subscripts the index of a Face, Edge, or Vertex. These are:

FACE LEVEL ATTRIBUTES:

FACEA[f]	the A coefficient of Face f
FACEB[f]	the B coefficient of Face f
FACEC[f]	the C coefficient of Face f
FACEK[f]	the K coefficient of Face f

Whenever these Attributes are accessed, all Faces on the Element are checked for planarity. If they are not, an error message is returned.

VERTEX LEVEL ATTRIBUTES:

VX[n]	the X coordinate of vertex n
VY[n]	the Y coordinate of vertex n
VZ[n]	the Z coordinate of vertex n

A Geometry is defined by the position of each Vertex. The Face level attributes are automatically computed from the Vertex information. A Shape is defined by specifying a Topology, followed by a Geometry, defined as a set of bindings to the vertex co-ordinates. Each X, Y and Z co-ordinate is bound to a value relative to the co-ordinate origin. Those not specified are defaulted to zero. The values can be specified as numerical expressions or Attributes. If a coordinate value evaluates to a single Attribute within the current Form or Copy, the binding is delayed until instantiation. Otherwise the coordinate will be bound to a literal value which is the value of the expression assigned.

In assigning Vertex coordinates, lists of subscripts may be defined in two different ways: either a simple list of subscripts or else a consecutive range may be defined by [n1 TO n2], indicating that n1, n2 are assigned, and all integer values between. If nonconsecutive values are desired, then [n1,n2,n3 TO n4] or any mix may be used. (In the consecutive form, n1 < n2 and in the nonconsecutive form, any order is acceptable.)

```
<> The shape of Cheops pyramid:
{ SHAPE = PYRAMID; VZ[1]= 400;
      VX[2,3]= -500; VX[4,5]= 500;
      VY[2,5]= 500; VY[3,4]= -500}

<> A shape definition procedure for a cuboid:
FORM PROCEDURE CUBOID(REAL L,W,H) =
      BFORM
      SHAPE = EXTRUDE(4);     !A hexahedron topology;
      VX[1,2,5,6]= L;
      VY[2,3,6,7]= W;
      VZ[4 TO 7]= H;
      END;

<> To build a pseudo-cylindrical column with N-gon section.
FORM PROCEDURE COLUMN(REAL N :RADIUS R :HEIGHT H) =
BFORM
SHAPE = EXTRUDE(N);
FOR I FROM 1 TO N DO
      BEGIN
      VX[I,I+N]= R*SIN(I*360/N);
      VY[I,I+N]= R*COS(I*360/N)
      END;
VZ[N+1,2*N]= H
END;
```

```
shape-spec ::= SHAPE = topo {;coord-ass }
coord-assign ::= coord-id subscript-list = n | statement
      A statement in the context of a Shape specification
      can contain coord-assignments embedded within it.
coord-id ::= VX | VY | VZ
subscript-list ::= "[" sub-range {,sub-range} "]" | "[" ALL "]"
sub-range ::= n [ TO n ]
```

3.8.4 ACCESSING AND ALTERING SHAPE INFORMATION

Accessing geometry attributes is done in the same way as all other Element Attributes. See Section 3.6. Modification of the geometry attributes of an existing Element, however, is limited. While the geometry attributes may not be directly updated, the vertex coordinates which evaluate to an Attribute are rebound each time they are accessed. Thus changing the value of an Attribute referred to by a geometry also alters the Shape. By initially defining a Shape's geometry in terms of a set of Attributes, any modification of a Shape is possible.

The Shape of different Copies of the Form can be varied by redefining the Attributes used in defining the Shape. Thus by having a unique LENGTH for each Copy of a Form, for example, each may have a different Shape.

```
<> ATTRIB REAL LENGTH,WIDTH,HEIGHT,NUMATCHES;
ATTRIB TEXT MANUF, COLOR;
FORM MATCHBOX = BFORM
      MANUF ← 'SUN MATCH INC';
      COLOR←'RED';
      NUMATCHES←100.0;
      LENGTH←5.6;
      WIDTH←2.6;
      HEIGHT←4.825;
      SHAPE = HEXA;
      VX[1,2,5,6] = LENGTH;
      VY[2,3,6,7] = WIDTH;
      VZ[4 TO 7] = HEIGHT
      END;
```

By setting up a Form in the above manner, each of its Copies may have a cuboid shape with different dimensions. The initial values assigned to the Attributes are the Form's default values.

```
COPY   MATCHBOX[3]   =   {\10,10,10;   LENGTH←7.3;   WIDTH←3.0};
COPY   MATCHBOX  =  {\0.5,100,100  *90;  WIDTH←3.0;  LENGTH←3.95};
```

3.8.5 SHAPE OPERATORS

New Shapes can also be created by combining existing Element Shapes, by sticking them together or cutting pieces out. The Shape operators that do this have as a value a new Shape, consisting of a Topology and Geometry. To be saved, this Shape must be assigned to a Form. Again the new Topology gets the name of the Form prefaced by "$".

COMB e1 WITH e2	The combination or union of the shapes.
LAP e1 WITH e2	The overlap or intersection.
CUT e1 FROM e2	The difference of e2 - e1

The local origin of any newly created Shapes is the local origin of the first operand. The Attributes of the input Shapes are not transferred.

```
FORM BLOB = { MATERIAL←'steel';
      COMBINE DOOR[3] WITH PIPE[14] };

shape-expr ::= COMBINE elem WITH elem | LAP elem WITH elem |
      CUT elem FROM elem
```

The LAP operation is the ultimate test for spatial conflicts. An example of its use will be presented later.

3.9 STANDARD ATTRIBUTES

There are a number of standard system-defined Attributes automatically associated with any Element. None of these may be explicitly modified by direct assignment, but are computed as results of operations on other parts of the Element. They include:

NUMCS	the number of Copies of the Form
TOPOL	the Topology used to describe the shape of an Element
NUMVERTS	the number of Vertices in an Element
NUMFACES	the number of Faces in an Element
NUMEDGES	the number of Edges in an Element
MAXX,MINX,MAXY,MINY,MAXZ,MINZ	*the largest and least* X, Y *and* Z *values* for the vertices in this Element

It is sometimes desireable to determine if a Form has a Shape. This may be determined by evaluating NUMVERTS; if zero, there is no Shape associated with the Element. Other system defined Element Attributes are defined in Sections 3.9. All system defined Attributes for an Element are listed in Appendix 5.

11-20-9

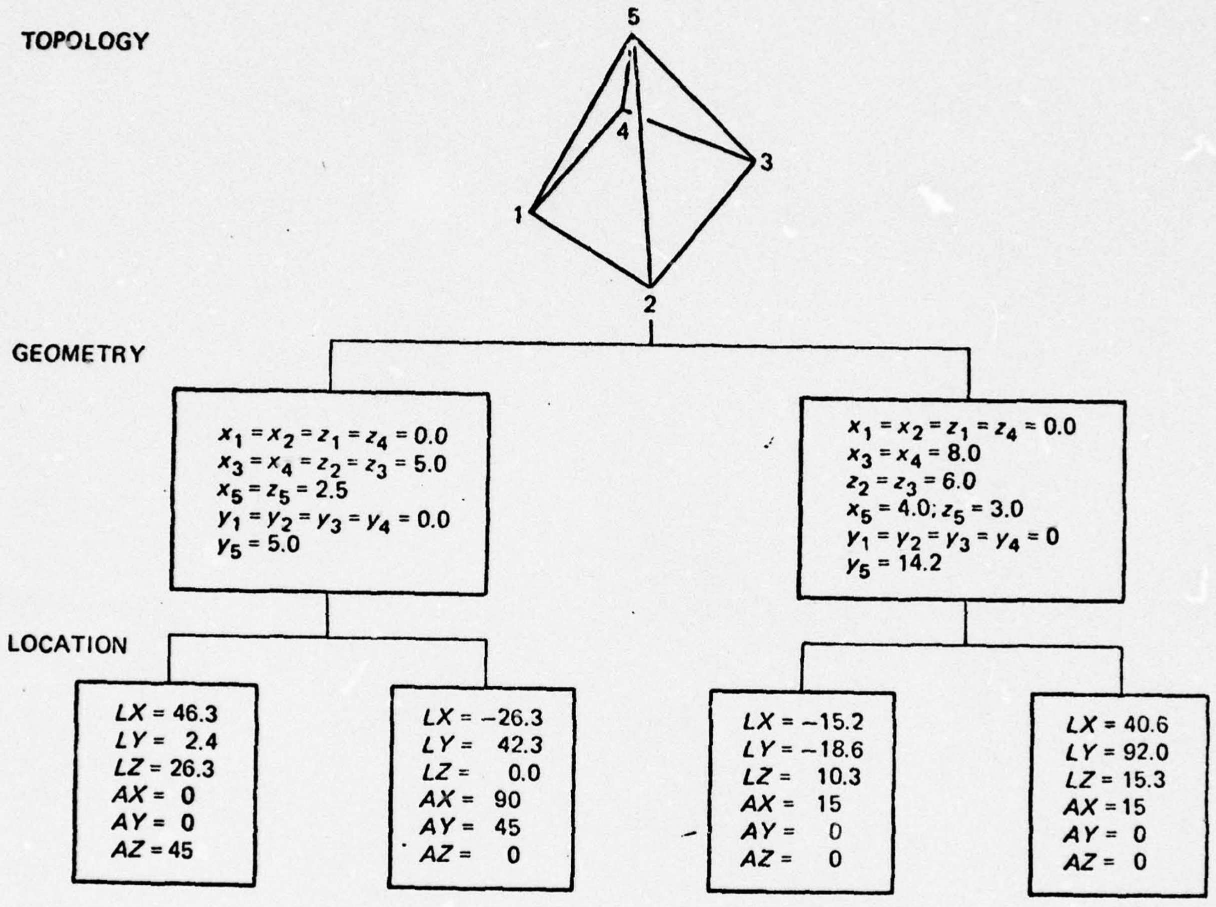


Figure 1: Three level hierarchy of topology, geometry and location used in GLIDE.

4 SETS

The Set record provides a way of referencing a number of Elements together so that they can be treated as a single entity. A Set can contain Elements and also other Sets. Thus multi-level hierarchies can be defined. A Set can contain no more than one reference to a Member, and attempts to enter duplicates are prevented. However, an Element may be both a member of a Set S1 and a member of S2 which is also a Member of S2; GLIDE will not recognize this duplication. All operators which take an Item for their arguments can operate on Sets.

The members of a Set continue to exist for as long as they are members, unless they are explicitly deleted, even after the scope in which they were created as locals has been exitted.

4.1 SET DEFINITION

A Set can be defined simply by a list of Elements and Sets enclosed between BSET and END (or, again { and }). A list of several Copies of the same Form can be specified concisely by listing the index range after the Form name. The keyword ALL means all the Copies of the Form.

An origin for rotation may be assigned to the Set as the first Element in the Set definition. If omitted the default is the co-ordinate origin.

```
<>
SET BOXES = { MATCHBOX[2]; SOAPBOX[1,3,7]; BOX[1 TO 25,27,30]};
SET TRASH = [\10,10,10; SOAPBOX[2,4];MATCHBOX[4 TO 17] };

set-defn ::= "{"[location ;] copy-range {;copy-range} "}"
        | BSET [location ;] copy-range {;copy-range} END
                copy-range ::= set | form-id [subscript-list] |
            set-id[subscript-list]
set-binding ::= SET name = set
set ::= set-id[index] | attribute | proced-call | set-defn |
      set-expr
```

4.2 COPYING A SET

It is possible to create Copies of a Set at different locations in the same way as Copies of a Form. They are similarly identified by subscripting the name of the original Set. Again the subscript may be specified by the user or supplied by the system.

In contrast to the Form, there are no consistency conditions on the Copies of a Set. Any Set of Set Copy may be composed of any Items. Copies may vary in a free way from the Sets they are initially made from.

```
<> COPY BOXES[1] = {\100,0, 100};

set-copy-decln ::= COPY set-id [index] = "{"location"}"
```

4.3 ACCESSING A MEMBER OF A SET

The nth member of a set may be accessed thus:

```
MEM[n] OF set
```

This is the nth member at the top level of the Set rather than the nth element going through subsets recursively, hence it may be an Element or another Set.

4.4 MODIFYING SETS

It is possible to modify single Elements within Sets in the usual way, by individually altering their location or other Attributes. It is also possible to insert and remove members of a set with standard functions PUT and TAKE.

```
PUT item IN set
TAKE item FROM set
```

TAKE gives an error message if the Element is not in the Set.

Note that Copies are made from the original Set-form as it is at that time, but subsequent alterations to the first Set will NOT affect any Copies previously derived from it.

4.5 STANDARD SET FUNCTIONS

NUMMEMS set gives the number of members of the Set.
ATOM item is a boolean function which returns TRUE if the Item is actually a single Element.

```
<> Examples:
procedure to test if a set contains an Item:
BOOL PROCEDURE MEMBER( ITEM E; :IN SET S) =
        BEGIN
        FORALL m IN S DO IF m EQL E THEN RETURN TRUE;
        RETURN FALSE
        END;

procedure to create union of two sets:
SET PROCEDURE UNION(SET S1, S2) =
        BEGIN ITEM S; S←{};
        FORALL m IN S1 DO PUT m IN S;
        FORALL m IN S2 DO PUT m IN S;
        S
        END;

        ! Set difference:
SET PROCEDURE DIFF(SET S1,S2)=
        BEGIN ITEM S; S←{};
        FORALL m IN S1 DO PUT m IN s;
        FORALL m IN S2 DO IF MEMBER m IN S1 THEN TAKE m FROM S;
        S
        END;
```

Procedure to enter attributes in inverted list form,
so that all Elements with a particular Attribute may be
accessed directly:

```
PROCEDURE COLLECT(ATTRIB SET ATT; :IN ELEMENT BODY; :IS SET VAL) =
        BEGIN
        BODY = {ATT ← VAL};
        PUT BODY IN VAL;
        END;
```

5.1 MOVE OPERATIONS

The relative placement of Items in space is accomplished with MOVE. It rotates an Item about its local origin, first about the X, then the Y then the Z axes, then translates the Item, either in absolute co-ordinates TO a new location, or relative to its current location BY an offset. Thus these two statements are equivalent:

```
<>MOVE BEAM[15] TO @ BEAM[15] \25,25;
MOVE BEAM[15] BY \25,25;
```

move ::= MOVE item TO location | MOVE item BY location

The origin for rotation of an Element's Shape can be moved by the operation MOVEO. The relative origin for rotation of Copies is an Attribute of the Form and so Copies of the same Form cannot differ in this respect. Thus MOVEO can apply to Forms and Sets, but not to their copies.

MOVEO item BY offset

A special form of MOVE operation is the negation of the transform, reverting the Shape to its location at the origin. This location is particularly useful for detailing operations. It consists of

ORIGIN item

To move an Element back from the origin to its world location, NORIGIN item is used. Notice that unintended use of this operation can result in permanent misplacement of an Element.

5.2 SEARCH OPERATIONS

This standard function returns the Set of Elements (possibly empty) which overlap or are contained within the specified Element.

FINDLAP elem

The use of the Findlap function is important for finding spatial conflicts and for determining topological relations between Elements. The routine below indicates how these issues might be handled in a particular application.

```
procedure for checking the spatial conflicts between one
Element and all others:
PROCEDURE WHAT.OVERLAPS( ELEMENT BODY) =
    BEGIN
    SET SS = FINDLAP BODY;
    WRITE 'These elements overlap it: ';
    FORALL TEMP IN SS DO
        BEGIN
        FORM INTER = {SHAPE = LAP BODY,TEMP};
        IF  NUMVERTS  OF  INTER  NEQ  0  THEN  WRITE  TEMP;
        END;
    END;
```

6 CONTROL STRUCTURES

There is no GO TO statement. All changes in flow of control are achieved by conditionals, loops, procedure calls and the EXIT, LEAVE and RETURN escape statements.

```
control-statement ::= conditional | loop | escape
escape  ::= leave | exit | return
loop ::= for-loop | while-loop | forall-loop
```

6.1 CONDITIONALS

```
conditional ::= IF b THEN statement [ELSE statement]
```

With the usual meaning: The ELSE part can be omitted. The if statement can be used as an expression, in which case it takes on the value and type of the statement which is executed, or 0 or null if the condition is false and the ELSE part is omitted. The "dangling else" ambiguity is resolved by binding it to the most recent IF THEN clause:

```
<> IF B1 THEN IF B2 THEN X ELSE Y;
is equivalent to:
IF B1 THEN (IF B2 THEN X ELSE Y);
not:IF B1 THEN (IF B2 THEN X) ELSE Y;
```

6.2 FOR LOOP

```
for-loop  ::=  FOR  name  [FROM  n1][TO  n2][BY  n3]  DO  statement
```

The continue condition is tested before each execution of the body and hence may be executed 0 or more times. The name is implicitly declared as a number for the scope of the body of the loop. The FROM, TO and BY clauses can each be omitted, with the following defaults:

n1 defaults to 1
n2 defaults to a very large number
n3 defaults to 1 if n2>n1 otherwise -1

6.3 WHILE LOOP

```
while-loop ::= WHILE b DO statement
```

If b is true execute the statement and repeat until b becomes false.

6.4 FORALL LOOP

```
forall-loop ::= FORALL name IN set DO statement
```

The name is implicitly declared as an Element with the scope of the loop body. The block is executed for each successive Element of the Set.

6.5 EXIT FROM A LOOP

exit ::= EXIT expression

This transfers control out of the current innermost loop to next statement after the end of the loop.

6.6 LEAVING A BLOCK:

leave ::= LEAVE [expr]

This passes control out of the inmost BEGIN-END block in which it occurs, and returns <expr>, if any, as the value of the block. <expr> must match the type of the block if any.

7 PROCEDURE DECLARATION

<> REAL PROCEDURE MAX(REAL a,b)=IF a GTR b THEN a ELSE b;

7.1 PROCEDURE TYPES

Procedures may be "functions" in that they return a value of a type which is specified in the declaration. The procedure type may be any scalar or record type, but not vectors or routines. It is omitted from the declaration if the procedure returns no value. The procedure body is an expression of the same type as the procedure, often being a BEGIN..END block.

7.2 PARAMETERS AND METHOD OF CALLING

The formal parameters are also typed, and must be explicitly declared at the beginning of the procedure definition. They can be of any type but cannot be procedures. Simple number and boolean type arguments are called by value, but all others types are called by reference, and the global version can be changed within the procedure.

7.3 ARGUMENT SEPARATORS

The parentheses enclosing arguments may be omitted in both calling and declaration of procedures. The separators for the parameters may either be commas, or they can be symbols of the same construction as names. These are declared in the formal declaration, preceded by ":" to identify them. This is to enable easy extensions of the language with command-style syntax for non-expert users. The appropriate choice of separators helps to identify the arguments where the procedure is called. If desired for conciseness commas can always be used in the call statement even if more expressive separators were defined.

```
< > SQRT n;
CUT Element FROM Element;
MOVE Element TO x, y, z;
REPEAT Element BETWEEN x, y, z AND x, y, z TIMES n;
```

7.4 RETURN STATEMENT

The return statement gives control back to the calling program, and returns the value of the associated expression, if any. The type of this expression must be compatible with the procedure type. If there is no return statement then the value of the procedure is the value of the last statement in the procedure body.

7.5 EXTERNALS

All user-defined names used within the procedure must be declared within it. This includes names which have already been declared at the outer level, and procedures whose type and number of arguments must also be declared using a similar syntax to the original declaration. These must be declared as external thus:

```
GET basic-decln
GET [type] PROCEDURE proced-id
      ( type arg1; : sep type arg2; :sep type arg3;... );
```

When the procedure is called the system links all the externals of the procedure to global names, at the same time as it links the formal parameters to actual values.

```
extern-proced-decln = GET proced-hdr-decln
proced-decln ::= proced-hdr-decln = statement
proced-hdr-decln ::= [type] PROCEDURE name [formals]

formals ::= ( basic-decln { sep-decln basic-decln} )
sep-decln ::= : name | ;
```

8. INPUT AND OUTPUT

8.1 ALPHANUMERIC OUTPUT

```
WRITE expr [ON device]
```

will output the value of the expression onto the device specified. The default device is the CRT diplay. The output may include text (in quotes) and other literals, as well as data from the database. The values of numbers, boolean and text values are output as literals. Record types are output by listing their names.

For more extensive and informative output regarding the structure of a file, the command

```
DUMP expr [ON device]
```

is available. In this case, record types are output in special formats: Forms are output with all Attribute names and their values, plus a list of all Copies. Copies are output as a listing of all Attributes, except for its Shape.

NEWLINE [ON device]
NEWPAGE [ON device] help to format output.

A space can be simply output thus: WRITE ' ';

8.2 ALPHANUMERIC INPUT

READREAL reads in a number from the default input device in either real or integer format, returning a real.

READTEXT reads a line of text up to but not including the carriage return, and returns it as a text variable.

8.3 GRAPHIC OUTPUT

Spatial Attributes, namely Shape and Location, of Elements can be output graphically onto the CRT display or plotter. To do this it is necessary to specify a view.

8.3.1 VIEW RECORD

View is a record type which can be declared and predefined. A View is defined in terms of a reference point, view point, cone of vision, whether a section is desired, and the kind of view. Kinds include perspective and orthographic. Sections are cut through the reference point normal to the direction of view.

```
<>
VIEW plan={ORTHO VIEWPT 0,0,10000}
VIEW front.elevation = {ORTHO VIEWPT 0,10000,0}
VIEW penthouse = { PERSP VIEWPT 1245, 2341, 110 REFPT 100,0,0 CONE 180}
view-decln ::= VIEW name = view-defn
```

```
view-defn ::="{" view-sort [SECTION]
      [VIEWPT n,n,n] [REFPT n,n,n] [CONE n] "}"
view-sort ::= PERSP | ORTHO
```

The defaults for omitted clauses are:

Kind of view:	Perspective. Not sectioned.
View point:	1000,1000,1000
Reference point:	0,0,0
Cone of vision:	60 degrees

8.3.2 DISPLAY COMMANDS

Conceptually all Items to be displayed are entered into a Display Set by the command:

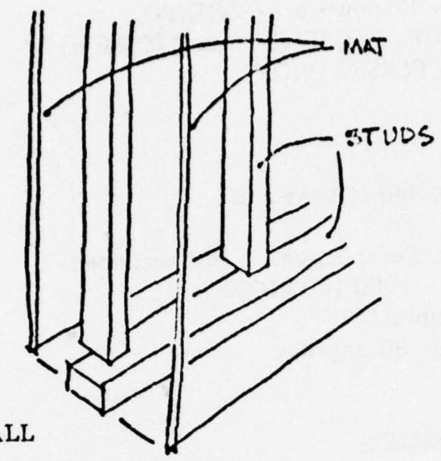
```
DISPLAY item [FROM view][ON device]
```

and are immediately diplayed from the view specified, if any. Default view is the last mentioned in a command. Default device is the last on mentioned (CRT or PLOTTER).

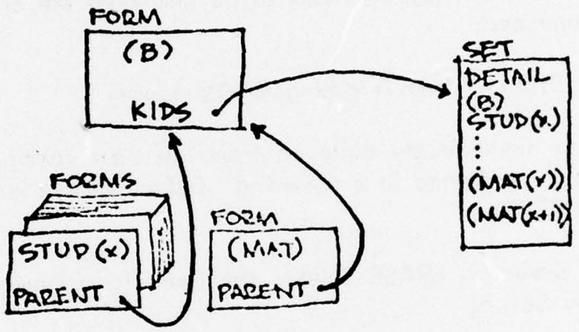
Similarly, ERASE deletes the Items from view by removing them from the Display Set, eg

```
ERASE item
```

EXAMPLE ONE:

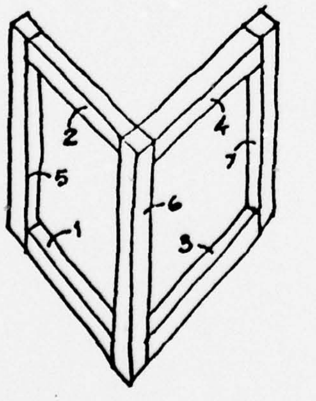


(a) LAYOUT OF STUDWALL



(b) Hierarchical organization of records created in the second version of STUDWALL. The first MEM of a set DETAIL is the PARENT, all others SIBLINGS.

EXAMPLE TWO:



$$\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
7 & & & \times & \times & & & \\
6 & \times & \times & \times & \times & & & \\
5 & \times & \times & & & & & \\
4 & & & & & & & \\
3 & & & & & & & \\
2 & & & & & & & \\
1 & & & & & & &
\end{array}$$

<u>POSSIBLE LAYOUT</u> <u>ADJACENCY MATRIX</u>

PART III: SOME EXAMPLE GLIDE PROGRAMS

The creation of a database describing a significant design, in almost any field, will require a considerable amount of GLIDE code, written over long periods of time. A faster design process will depend on a library of powerful Glide subprocedures.

In the following examples, we try to show how some important design issues can be addressed using the language.

1. AUTOMATIC DETAILING

Most organizations work out standard ways of detailing certain parts of a design. These may be large units such as a hospital room or small details such as a joint or fitting. A design language should allow definition of these standard responses to a specific context and allow them to be applied automatically. The following procedures provide one example of this.

```
PROCEDURE STUDWALL(ELEM B, STUD, MAT; REAL SPACE)=
```

!..This procedure details a rectangular slab B as a stud wall, using two predefined Forms, STUD and MAT. Copies of STUD are created spaced SPACE units apart. The studs are covered with a sheet of material MAT on each sides. The procedure assumes that the original definition of all wall Forms is a rectangle oriented along the X-axis. The same is true for STUD. MAT is expected to include two attributes, LENGTH and WIDTH, which determines the covering size. The wall detail is centered on the centerline of the original rectangle, B;

```
    BEGIN
    ATTRIB LENGTH,WIDTH;
    SET TEMSET;
    REAL TX,TY,TZ,LEN,HT;

    ! Store  a copy of B;
    ELEM B0 ← COPY B={@B};
    ! All attributes will refer to B
    B = {   TX ← MINX;
            TY ← MINY;
            TZ ← (MAXZ-MINZ)/2-1.75;
            HT ← (MAXY-MINY)-1.5;
            LEN ← (MAXX-MINX)  ;
            !..create plate and runner;
            PUT  COPY    STUD  =  {\TX,TY,TZ;  LENGTH←LEN  }  IN  TEMSET;
            PUT  COPY    STUD  =  {\TX,HT,TZ;  LENGTH←LEN  }  IN  TEMSET;

            !..create studs;
            LEN ← HT-1.5;
            TY ← TY+1.5  ;
            FOR  J FROM MINX+1.5 TO MAXX  BY SPACE DO
                  PUT COPY STUD= {\J,TY,TZ *0,0,90; LENGTH←LEN } IN TEMSET;
            !..place end stud;
            PUT  COPY  STUD  =  {\MAXX  OF  B,TY,TZ  *0,0,90;  LENGTH←LEN  }
                  IN TEMSET;
```

```
        !..attach surface materials;
        TZ ← TZ+3.5;
        B = { LN ← (MAXX-MINX);     HT ← (MAXY-MINY) };
        PUT   COPY   MAT=   {\TX,TY,TZ;   LENGTH←LEN;WIDTH←HT    }
              IN TEMSET;
        TZ ← TZ-3.5;
        PUT COPY MAT= {\MAXX OF B,TY,TZ *0,180,0; LENGTH←LEN; WIDTH←HT }
              IN TEMSET;

        !..move rectangle and all its parts back into location;
        MOVE TEMSET BY @B0;
        DELETE {B,B0};
    END;
```

Notice that the temporary set TEMSET that groups all new elements allows them to be moved together, rather than one at a time. At the end of the procedure, B is deleted from the database. This procedure is facilitated by the ORIGIN operation and the MIN and MAX attributes.

This detailing procedure is easily extended so as to allow maintaining the hierarchical relations between sets of Elements. That is, instead of deleting the original rectangle, a designer may wish to retain it, in that it has many important uses. First, it allows crude drawings without the detail of each stud. It also aggregates data for the complete wall. If one desires the structural or acoustical performance of the wall detail, the enclosing rectangle is an appropriate single unit for storing that information, rather than with one of its components. The capability of maintaining hierarchical relations is an important capability in design, whether manual or automated.

```
PROCEDURE STUDWALL(ELEM B, MAT, STUD; REAL SPACE)=
    !..This extended procedure makes a Set DETAIL with the details in it.
The B has a set attribute KIDS which points to set DETAIL that contains the
elements which comprise it. Each component has an attribute PARENT of type ELEM
which points to its parent, B;

    BEGIN

    GET SET DETAIL;
    ATTRIB ELEM PARENT, KIDS;
    ATTRIB REAL LENGTH,WIDTH;
    REAL TX,TY,TZ,LEN,HT;

    ! Store  a copy of B;
    ELEM B0 ← COPY B={@B};
    ORIGIN B;


    !..create plate and runner;
    B = { KIDS←COPY DETAIL={};        ! creaate empty Set
          TZ ← (MAXZ-MINZ)/2-1.75;
          HT ← (MAXY-MINY)-1.5;
          LEN ← (MAXX-MINX) };
```

```
        PUT   STUD={\MINX,MINY,TZ; LENGTH←LEN} IN KIDS;
        PUT   STUD={\TX,HT,TZ; LENGTH←LEN} IN KIDS;
        !..create studs;
        LEN ← HT-1.5;
        TY ← TY+1.5;
        FOR  X FROM MINX+1.5 TO MAXX BY SPACE DO
             PUT  COPY  STUD=  {\X,TY,TZ  *0,0,90;LENGTH←LEN}  IN  KIDS;

        TX ← MAXX ;
        PUT  CQPY STUD = {\TX,TY,TZ *0,0,90;
               LENGTH←LEN };

        !..attach surface materials;
        TZ ← TZ+3.5;
        B = { LN ← (MAXX-MINX);
              HT ← (MAXY-MINY)  };
        PUT    COPY  MAT=  {\TX,TY,TZ;  LENGTH←LEN;WIDTH←HT}  IN  KIDS;
        TX←MAXX;
        TZ ← TZ-3.5;
        PUT COPY MAT= {\TX,TY,TZ *0,180,0; LENGTH←LEN;WIDTH←HT} IN KIDS;

        ! Put in references up the hierarchy.
        FORALL E IN KIDS DO E={PARENT←B};
        !..relocate all wall components;
        MOVE DETAIL[SN] BY @ BO;
        DELETE BO};
   END;
```

Access to the components of the wall B can be made through use of:

```
   FORALL S IN KIDS OF B DO  statement
```

2. BUILDING TOPOLOGICAL STRUCTURES

An important use of GLIDE is the deriving of information needed for analysis programs. Of the data that must be generated, possibly the most critical are the different topological relations between Elements, such as the structural grid or mechanical system tree. Below, we indicate how topological relations may be computed. The capability developed is a general procedure that generates anadjacency matrix of all Elements within a Set that are adjacent to one another. Thus if the Set is of structural Elements, the result will be the structural grid. If the Set includes all mechanical equipment, then the result will be a matrix with the tree of all mechanical equipment adjacencies.

Several service procedures are introduced first.

```
BOOL PROCEDURE  TOLERANCE(REAL A,B,C) = ((A+C GTR B) AND (A-C LSS B));

PROCEDURE  EXTFACE(ELEM BODY; REAL F; REAL VECTOR C[6]) =
!        This procedure  returns the  coordinates of  the enclosing  box
around  any  face  and  is  useful  in  determining  if  two  faces  overlap;

     BEGIN
     REAL A,B;

     FORM BODY = { B←A←V.FACE(F,0);
          C[4]←C[1] ← VX[A];
          C[5]←C[2] ← VY[A];
          C[3]←C[6] ← VZ[A];
          WHILE ((B ← V.FACE(F,B)) NEQ A) DO
               BEGIN
               IF C[1] GTR VX[B] THEN C[1] ← VX[B];
               IF C[4] LSS VX[B] THEN C[4] ← VX[B];
               IF C[2] GTR VY[B] THEN C[2] ← VY[B];
               IF C[5] LSS VY[B] THEN C[5] ← VY[B];
               IF C[3] GTR VZ[B] THEN C[3] ← VZ[B];
               IF C[6] LSS VZ[B] THEN C[6] ← VZ[B];
               END;
     } ;
     RETURN
     END;


BOOL PROCEDURE ADJACENT(ELEM A,B) =
     !..this  procedure  tests  the   adjacency  between  elements  A  and  B
and returns TRUE if any of their faces are coincident;

     BEGIN
     EXTERNAL EXTFACE;
     REAL VECTOR AF[6],BF[6];

     FOR I1 FROM 1 TO (NOFACES OF A) BY 1  DO
          BEGIN
          EXTFACE(A,AF);
          FOR I2 FROM 1 TO (NOFACES OF B) BY 1 DO
               BEGIN
               EXTFACE(B,BF);
          !..use the boxtest to see if faces overlap;
               IF AF[1] GTR BF[4] OR BF[1] GTR AF[4] OR
                  AF[2] GTR BF[5] OR BF[2] GTR AF[5] OR
                  AF[3] GTR BF[6] OR BF[3] GTR AF[6] THEN
                     EXITLOOP;
          !..test if the faces align;
               IF   TOLERANCE(FACEA(I1)  OF  A,FACEA(I2)  OF  B)  AND
                  TOLERANCE(FACEB(I1)  OF  A,FACEB(I2)  OF  B)  AND
                  TOLERANCE(FACEC(I1)  OF  A,FACEC(I2)  OF  B)  AND
                  TOLERANCE(FACEK(I1)  OF  A,FACEK(I2)  OF  B)  AND
               THEN RETURN TRUE;
               END;
          END;
     RETURN FALSE
     END;
```

!..this procedure builds an adjacency matrix, ADJ,betweeen the Elements in the Set S. ADJ is the vector in which adjacencies are stored. N is the dimension of ADJ. It should be equal to (NUMEM * (NUMEM-1))/2, where NUMEM is the number of members of S;

```
PROCEDURE GRAF(SET S; BOOL VECT ADJ[N]) =
        FOR I1 FROM 1 TO N-1 BY 1 DO
                FOR I2 FROM I1+1 TO N  DO
                        ADJ[I1*(I1-1)/2+I2] ←
                                ADJACENT(MEM(I1,S), MEM(I2, S));
```

3. ANALYSES OF SHAPES

An important aspect of some design operations is the analysis of shapes and the properties of shapes. Examples include computation of the amount of concrete required for a pour or the length of run of a pipe or duct. These operations require the ability to access and respond to different conditions encountered within the topology of a shape.

Below, we provide one meaningful example, an algorithm for computing the volume of any shape.

REAL PROCEDURE DET(REAL T,V1,V2,V3; ELEM B) = !this procedure computes the projected area of a triangle defined by the three vertex IDs V1,V2,V3 onto the plane denoted by XYZ, where XYZ=1 means X, XYZ=2 means Y and XYZ=3 means Z plane. when the points are taken clockwise, the area sign is negative;

```
        BEGIN
        REAL A;
        B = {      IF (XYZ EQL 3) THEN A←VX[V1]*VY[V2]-VX[V1]*VY[V3]-
                   VY[V1]*VX[V2]+VY[V1]*VX[V3]
                   +VX[V2]*VY[V3]-VY[V2]*VX[V3]
             ELSE  IF (XYZ EQL 2) THEN A←VZ[V1]*VX[V2]-VZ[V1]*VX[V3]-
                   VX[V1]*VZ[V2]+VX[V1]*VZ[V3]
                   +VZ[V2]*VX[V3]-VX[V2]*VZ[V3]
             ELSE  A←VY[V1]*VZ[V2]-VY[V1]*VZ[V3]-
                   VZ[V1]*VY[V2]+VZ[V1]*VY[V3]
                   +VY[V2]*VZ[V3]-VZ[V2]*VY[V3]●};
        RETURN A/2
        END;
```

REAL PROCEDURE AREA(REAL F; ELEM BODY)= ! this procedure computes the area of face F on body BODY. It chooses the plane of projection most parallel to the face, then corrects the projected area by the cosin of the angle between the face and the projection plane;

```
        BEGIN
        REAL A,B,C,D,ANGLE,PLANE;

!... compute area of projection;
        BODY  =  {IF  FACEA(F)  GTR  FACEB(F)  THEN  PLANE←1  ELSE  PLANE←2;
                IF FACEC(F) GTR (IF PLANE EQL 1 THEN FACEA(F)
                     ELSE FACEB (F) )
                     THEN PLANE←3;
                B←V.FACE(F,0);
                C←V.FACE(F,B);
                D←V.FACE(F,C);
                A←DET(PLANE,B,C,D,BODY);

                WHILE BEGIN
                        C←D;
                        D←V.FACE(F,D);
                        ( D NEQ B )
                        END
                DO    A←A+DET(PLANE,B,C,D,BODY);
!... compute angle of projection;
                B←FACEA(F)*FACEA(F)+FACEB(F)*BACEB(F)+FACEC(F)*FACEC(F);
                IF PLANE EQL 1 THEN D←FACEA(F) ELSE
                        IF  PLANE  EQL  2  THEN  D  ←  FACEB(F)  ELSE  D  ←  FACEC(F);
                ANGLE←D/(SQRT(B))};
        RETURN A/ANGLE
        END;


REAL PROCEDURE VOLUME(ELEM B) =
! this procedure computes the volume of  element B;
        BEGIN
        REAL DIST,J,AREA1,VOL,V;
        V←0;
        VX[V]←VY[V]←VZ[V]←0;
        AREA1←0; VOL←0;

        FOR J FROM 1 TO NUMFACES(B) DO
                B = {   DIST←FACEK(J);
                        AREA1←AREA(J,B);
                        VOL←VOL+(DIST*AREA)};
        RETURN VOL/3
        END;
```

In the examples provided, we have attempted to indicate the kinds of design operations easily defined within GLIDE. The full extent of the strengths and weaknesses of the language will only be known after extensive experience with this and other design oriented languages.

APPENDICES:

APPENDIX 1 BNF Conventions used:

1. Non-terminal symbols are represented by single or hyphenated words in lower-case, without the <> delimiters.

    eg statement, proced-call, form-defn

2. Terminal symbols (keywords)are in upper-case:eg: IF, THEN, BEGIN, or are special characters :, *, +, <, !

3 In the BNF the symbol "name" is used to signify a hitherto undeclared name, to be used in a declaration, or a name of any type. Once they have been declared, names of various types are referred to thus:

    num-id bool-id text-id topo-id form-id item-id set-id view-id routine-id command-id attrib-id elem-id

4. | as usual seperates alternative strings.

    eg x ::= a |b c | d

5. [ ] enclose optional strings

    eg a ::= b[c] means a ::= b | b c

6. { } enclose strings repeated 0 or more times.

    eg: a ::= b {c} means a ::= b | a c

7. Literal terminal symbols {}[] occurring in the syntax are enclosed in " " to distinguish them from the meta-symbols defined above.

8. Certain parts of the syntax are highly context-dependent. For example a statement within certain definition blocks can contain certain operations which are illegal in other contexts. Such a context is valid no matter how deeply a statement is nested inside the definition block:

A Euler operations can only occur within a Topology definition block.

B Attribute assignments can only occur within a Form or Copy definition block.
C Co-ordinate bindings can only occur within a Shape definition block.

D In almost all contexts an expression must be of a particular type, eg the operands of an assignment must match. In some cases this constraint is not inherent in the syntax as specified in the BNF, but merely noted alongside.

APPENDIX 2: TABLE OF INFIX OPERATORS WITH PRECEDENCE.

Preced.	Operators
1	← =
2	OR
3	AND
4	LSS LEQ NEQ EQL GEQ GTR
5	+ -
6	* /
7	↑ OF

APPENDIX 3: RESERVED WORDS AND SPECIAL SYMBOLS.

! # & ' ( ) = \ @ { } [ ] + * ; : . , / ←

AND OR OF FOR FORALL WHILE IF BEGIN END BTOPO BFORM BSET RETURN EXIT DELETE SHAPE PROCEDURE GET GLOBAL ATTRIB TOPO FORM COPY ITEM ELEM SET REAL BOOL TEXT VIEW ALL FROM TO BY DO THEN ELSE ORTHO PERSP VIEWPT REFPT CONE SECTION

APPENDIX 4: STANDARD FUNCTIONS AND ATTRIBUTES.

Standard attributes:

SHAPE TOPOL VX VY VZ CX CY CZ AX AY AZ NUMVS NUMFS NUMCS NUMELS MAXX MINX MAXY MINY MAXZ MINZ E.FACE V.FACE F.EDGE V.EDGE F.VERT E.VERT FACEA FACEB FACEC FACEK

Standard functions and operations:

MOVE MOVEO ORIGIN NORIGIN CMV CFE MVE MFE FETCH CLOSE COMBINE CUT LAP FINDLAP WRITE DUMP NEWLINE NEWPAGE READREAL READTEXT MEMBER